PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Chitra G M, Neeta Ann Jacob
Computer Science and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Operators

Chitra G M, Neeta Ann Jacob
Department of Computer Science and Engineering

Operators

- **An operator is a symbol that represents an operation that may be performed on one or more *operands***
- For example, the + symbol represents the operation of addition
- **An operand is a value that a given operator is applied to,** such as operands 2 and 3 in the expression 2 + 3

Operators

Arity or Rank:

- A **unary operator** operates on only one operand, such as the negation operator in the expression: **- 12**

- **A binary operator operates on two operands**, as with the addition operator: **2+3**

Operators

Arithmetic Operators

Operator	Expression	Name
-	-x	Negation
+	x + y	Addition
-	x - y	Subtraction
*	x * y	Multiplication
**	x ** y	Exponentiation
/	x / y	Division
//	x // y	Truncation Division
%	x % y	Modulus

Operators

Division

Python provides two forms of division:

- **"True" division** is denoted by a single slash, **/**
  Thus, **25 / 10 evaluates to 2.5**
- **Truncating division** is denoted by a double slash, **//** providing a truncated result based on the type of operands applied to

When both operands are integer values, the result is a truncated integer referred to as **integer division**.

When as least one of the operands is a float type, the result is a **truncated floating point**.

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Operators

	Operands	result type	example	result
/ Division operator	int, int	float	`7 / 5`	`1.4`
	int, float	float	`7 / 5.0`	`1.4`
	float, float	float	`7.0 / 5.0`	`1.4`
// Truncating division operator	int, int	truncated int ("integer division")	`7 // 5`	`1`
	int, float	truncated float	`7 // 5.0`	`1.0`
	float, float	truncated float	`7.0 // 5.0`	`1.0`

Modulus Operator

Modulus operator (%) **gives the remainder of the division of its operands**, resulting in a cycle of values

Modulo 7		Modulo 10		Modulo 100	
0 % 7	**0**	0 % 10	**0**	0 % 100	**0**
1 % 7	**1**	1 % 10	**1**	1 % 100	**1**
2 % 7	**2**	2 % 10	**2**	2 % 100	**2**
3 % 7	**3**	3 % 10	**3**	3 % 100	**3**
4 % 7	**4**	4 % 10	**4**	.	.
5 % 7	**5**	5 % 10	**5**	.	.
6 % 7	**6**	6 % 10	**6**	96 % 100	**96**
7 % 7	0	7 % 10	**7**	97 % 100	**97**
8 % 7	1	8 % 10	**8**	98 % 100	**98**
9 % 7	2	9 % 10	**9**	99 % 100	**99**
10 % 7	3	10 % 10	0	100 % 100	0
11 % 7	4	11 % 10	1	101 % 100	1
12 % 7	5	12 % 10	2	102 % 100	2

What Is an Expression?

An expression is a combination of symbols that evaluates to a value.

Expressions, most commonly, consist of a combination of operators and operands,

$$4 + (3 * k)$$

An expression **can also consist of a single literal or variable**
Thus, 4, 3, and k are each expressions

Expressions that evaluate to a numeric type are called arithmetic expressions

Operator Precedence

- Determines the order of evaluation
- Consider the following expression:

4+3*5

  - There are two possible ways in which it can be evaluated

4 + 3 * 5 → 4 + 15 → **19** 4 + 3 * 5 → 7 * 5 → **35**

- **Each programming language has its own rules for the order that operators are applied, called operator precedence**

Operators

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right

In the table, higher-priority operators are placed above lower-priority ones.

4 + 3 * 5 → 4 + 15 → 19

In our example, therefore, if the addition is to be performed first, parentheses would be needed,

(4 + 3) * 5 → 7 * 5 → 35

Operator Precedence

As another example,

```
4 + 2 ** 5 // 10  →  4 + 32 // 10  →  4 + 3  →  7
```

Operator precedence guarantees a consistent interpretation of expressions

It is good programming practice to use parentheses even when not needed

```
4 + (2 ** 5) // 10
```

Operator Associativity

If more than one operator with the same level of precedence exists, **association** indicates the order of evaluation

For operators that follow the associative law (such as addition) the order of evaluation doesn't matter

$(2 + 3) + 4 \rightarrow 9$ $\quad$ $2 + (3 + 4) \rightarrow 9$

Division and subtraction, however, do not follow the associative law,

(a) $(8 - 4) - 2 \rightarrow 4 - 2 \rightarrow 2$ $\quad$ $8 - (4 - 2) \rightarrow 8 - 2 \rightarrow 6$

(b) $(8 / 4) / 2 \rightarrow 2 / 2 \rightarrow 1$ $\quad$ $8 / (4 / 2) \rightarrow 8 / 2 \rightarrow 4$

Operators

Operator Associativity

operator associativity defines the order that it and other operators with the same level of precedence are evaluated

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right

Therefore, 23**2**

2 ** (3 ** 2) → **512** **and not** (2 ** 3) ** 2 → **64**

Boolean Expressions

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python

A **Boolean expression** is an expression that evaluates to a Boolean value

Boolean expressions are used to denote the conditions for selection and iterative control statements

Operators

Relational Operators

Used to compare two values.

Relational expressions are a type of **Boolean expression**, since they evaluate to a Boolean result

Relational Operators	Example	Result
== equal	10 == 10	True
!= not equal	10 != 10	False
< less than	10 < 20	True
> greater than	'Alan' > 'Brenda'	False
<= less than or equal to	10 <= 10	True
>= greater than or equal to	'A' >= 'D'	False

Operators

Relational Operators

- **Simple comparison**

```
10 == 10        True
3 > 2           True
```

- **Cascading comparison**

a op1 **b** op2 **c** is the same as (a op1 b) and (b op2 c)

3 > 2 > 1 is the same as **(3>2)** and **(2>1)** **True**

Relational Operators

- **String comparison:**

Compares the corresponding characters based on the ASCII value

“ cat ” > “car”	# True # “ t ” > “ r ”
“ cat ” > “ cattle"	# False : Second string is longer
“ cat ” == “Cat”	# False : “ C ” < “ c ”
“ apple ” > “z”	# False :Comparison not based on the length
“ zebra ” > “abcedefgh”	# True “ z ” > a ”

Relational Operators

- **List comparison:**

Rules are same as that of string - compare the corresponding elements until a mismatch or one or both ends

[10, 20, 30] > [10, 25] # False 20 > 25 is false

[(10, 20), "abcd"] >[(10, 20), "abcc"] # True d of abcd > last c of abcc

Operators

Membership Operators

- These operators can be used to determine if a particular value occurs within a specified collection of values.

Membership Operators	Examples	Result
in	10 in (10, 20, 30)	True
	red in ('red','green','blue')	True
not in	10 not in (10, 20, 30)	False

- The membership operators can also be used to check if a given string occurs within another string

Boolean (Logical) Operators

- Boolean algebra contains a set of **Boolean (logical) operators**
- Denoted by **and**, **or**, and **not**.
- These logical operators can be used to construct arbitrarily complex Boolean expressions

x	y		x and y	x or y	not x
False	False		False	False	True
True	False		False	True	False
False	True		False	True	
True	True		True	True	

Boolean (Logical) Operators

- **False Values**: 0 , ‘’ (Empty String), [] , {} , () (Empty Collections)
- **True Values:** non – Zero numbers , Non Empty String, Non Empty Collections

Short Circuit Evaluation

- logical and, if the first operand evaluates to false, then regardless of the value of the second operand, the expression is false
- logical or, if the first operand evaluates to true, regardless of the value of the second operand, the expression is true.
- Python interpreter does not evaluate the second operand when the result is known by the first operand alone
- This is called **short-circuit (lazy) evaluation**

Operators

Operator Precedence and Boolean Expressions

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right
<, >, <=, >=, !=, == (relational operators)	left-to-right
not	left-to-right
and	left-to-right
or	left-to-right

Bitwise Operators

Operations are performed at the bit level

- & => AND : result is 1 if the corresponding bits are one
- | => OR : result is 1 if even one of the bits is one
- ^ => Exclusive OR : result is 1 if and only if one of the bits is 1
- << => LEFT SHIFT : multiply by 2 for each left shift
- >> => RIGHT SHIFT : divide by 2 for each right shift
- ~ => ONE'S COMPLIMENT : change 0 to 1 and 1 to 0

Operators

- & AND

```
a = 5          # 0101
b = 6          # 0110
c = a & b      # 0100  (4)
```

- | OR

```
a = 5          # 0101
b = 6          # 0110
c = a | b      # 0111  (7)
```

- ^ XOR

```
a = 5          # 0101
b = 6          # 0110
c = a ^ b      # 0011  (3)
```

Operators

- >> Right Shift

a = 5 # 0101

b = a >> 2 # 0001

Working:

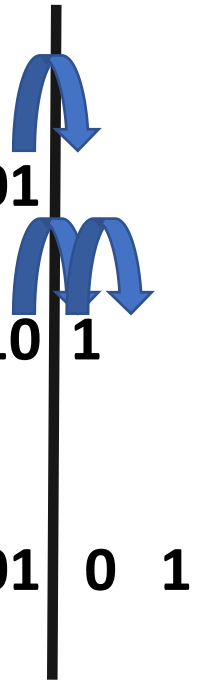
a = 5 **#0101**

Shift 1 bit to the right **#0010 1**

a = 2

Shift 1 bit to the right **#0001 0 1**

a = 1



Operators

- << Left Shift

```
a = 5          # 0101
b = a << 2     # 010100
```

Working:

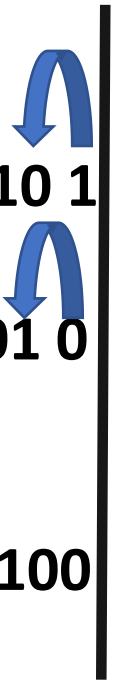
a = 5 **#010 1**

Shift 1 bit to the left **#0101 0**

a = 10

Shift 1 bit to the left **#010100**

a = 20



Identity Operators

Checks if the operands on either side of the operator point to the same object or not

Denoted by **is** and **is not**

Example:

10 is 10 **#True**

10 is not 11 **#True**

Assignment / Shorthand Operators

Combines arithmetic and assignment operators

Operator	Expression	Short Hand
+= (Addition)	a = a + b	a += b
-= (Subtraction)	a = a - b	a -= b
*= (Multiplication)	a = a * b	a *= b
/= (Division)	a = a / b	a /= b
//= (Truncation Division)	a = a // b	a //= b
%= (Modulus)	a = a % b	a %= b
**= (Exponentiation)	a = a ** b	a **= b

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Operators

1. Arithmetic Operators **(+ , - , * , / , // , % , **)**
2. Relational operators **(== , != , < , <= , > , >=)**
3. Logical or Boolean operators **(and , or , not)**
4. Membership operators **(in , not in)**
5. Bitwise operators **(& , | , ^ , >> , << , ~)**
6. Identity operators **(is , is not)**
7. Assignment operators / shorthand operators

   (+= , -= , *= , /= , //= , %= , **=)

THANK YOU

Chitra G M , Neeta Ann Jacob
Department of Computer Science and Engineering

chitragm@pes.edu
+91 9900300411

neetajacob@pes.edu
+91 9844820045